Martín-Pérez, J., Bernardos, C.J. (2018). *Multi-domain VNF mapping algorithms*. Paper submitted in 2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB), Valencia.

DOI:10.1109/BMSB.2018.8436765

# Multi-domain VNF mapping algorithms

J. Martín-Pérez
IMDEA Networks Institute
Universidad Carlos III de Madrid
Madrid, Spain
Email: jorge.martin@imdea.org

Carlos J. Bernardos
Universidad Carlos III de Madrid
Madrid, Spain
Email: cjbc@it.uc3m.es

***Abstract*—5G technologies are taking benefit of the Network Function Virtualization to achieve more flexible deployments. This new paradigm allows the resource sharing between operators in federated environments thanks to the decomposition of services into virtual network functions connected together composing a "service function chain". This work proposes algorithms to solve the placement of such chains in federated multi-domain scenarios satisfying imposed restrictions in terms of resource sharing. Algorithms run on top of an implemented simulator for federated scenarios where multiple operators are involved. Two of our proposed solutions reach $O(N)$ running times in certain scenarios. Our results also show that we achieve acceptance ratios very similar to those obtained using a tabu meta-heuristic implementation.**

***Index Terms*—multi-domain, VNF, mapping, heuristic**

## I. Introduction

Network Services (NS) like video streaming are usually made up of several components (firewalls, video optimizers, parental control, etc.) that form what is known as a Service Function Chain (SFC).

Such SFC is a set of Network Functions (NF) linked together to provide a service. Network Function Virtualization (NFV) technologies enable the virtualization of the NFs, obtaining Virtual Network Functions (VNFs), to provide more flexible deployments depending on the demand and requirements imposed by the Service Providers (SPs) that instantiate them.

In 5G technologies, NFV has led to an arising idea of federated multi-domain environments where several operators and SPs can share resources using virtualization layers. Infrastructure owners can let other entities use their servers and links, and make agreements to decide how they want to offer part of the resources to other entities in the multi-domain federation.

In this work we present algorithms to map NS requests on top of a federated multi-domain environment. We analyze and implement existing algorithms in the literature, variations of them, and also propose novel ones. Every algorithm studied in this work tries to achieve the lowest service delay, and to allocate the minimum resources necessary for the deployment.

To test the algorithms' performance we have implemented a simulator that generates federated multi-domain graphs and SFCs.

The remainder of this document is organized as follows. We analyze the related work in Section II. Section III explains the multi-domain graphs we can generate in our simulator. Section IV provides details of the mapping algorithms used to do the NS requests placement on top of the generated graphs. A stress test of the implemented algorithms is shown in Section V. Section VI finishes the paper with some conclusions and future work.

## II. Related work

The Virtual Network Functions mapping problem has been widely studied in the recent years. Several works are putting a big effort into the search of optimal solutions for the NS mapping problem, which is an $NP$ hard problem [1]. Some approaches rely on Integer Linear Programming (ILP) techniques to achieve optimal solutions, while others solve the problem with heuristic algorithms that reach solutions in feasible computational times.

Works like [2] and [3] use ILP and Mixed ILP (MILP) techniques to perform the NS placements. [2] focuses on a relaxation of the initial ILP formulation to solve the VNF placement and routing problem, while [3] creates an MILP-based heuristic to solve the problem using local branching.

There is a considerable number of articles which focuses on how to solve the problem with heuristics due to their short running times. Among these proposals, works like [1] use the simulated annealing algorithm to deploy the VNFs in the middle of traffic flows, so the deployments are optimal. Others use matrices and eigenvalues [4] to discover deployments that minimize the costs, or well known algorithms as Dijkstra to reduce the number of SDN flows [5]. Breadth First Search (BFS) based algorithms [6], Genetic algorithms [7], greedy heuristics [8], and even the tabu meta-heuristic algorithm [9] have been used as well in the VNF mapping problem. The latest has been modified in this work to minimize the end-to-end delay of the deployed SFC.

Works mentioned above do not address the mapping in federated scenarios, but projects as the 5GEx [10] are currently studying these solutions. [11] and [12] show some of the mapping algorithms ideas used in the 5GEx project.

The DFS (Depth First Search) is novel in the mapping problem, and we chose it to directly reach the leaves of our graphs (the servers). This paper proposes modifications in the DFS and BFS to boost up the mapping performance in terms of

running time; and a tabu algorithm based in [9] that minimizes the end-to-end mapping delay.

## III. Multi-domain graphs

To address the multi-domain environment we generate a federation graph representing every SP fat-tree data center [13] and the gateways that interconnect them (see Fig. 1).

We understand as federation the sharing of resources among the SPs present in the graph. This means that a SP within the federation can use servers and links present in the fat-trees of other SPs.

Two parameters can be tunned in the generation of the federation graph:

> $mesh\_degree$: it can take real values in the interval $(0, 1)$ and it controls how interconnected are the SP's gateways mesh. $mesh\_degree = 1$ means a full mesh, $mesh\_degree = 0$ means that gateways are connected in a circular graph.
> $k$: this parameter is used to specify the fat-tree size. More specifically it determines the number of pods, a set of switches and servers connected, in the fat-tree. In Fig. 1 each fat-tree is of size $k = 4$.

To verify the proper generation of the federation graph, we rely on a property of the fat-trees guaranteeing that servers inside the data center can be reached in 2, 4 or 6 hops; depending on how far they are between them. Hence checking the number of shortest paths of length 2, 4 and 6; is enough to verify the correct creation.

Once the simulator checks the proper generation of the federation graph, the next step performed is the sharing of resources. This involves deciding the amount of computational and link resources that each SP offers to other SPs in the federation.

This sharing is done following these steps:

1) Every SP decides which pods of its fat-tree are shared with other SPs.
2) Inside the pods' servers, the SP determines the amount of CPU, memory and disk that is allocated for the SPs that have access to them.
3) Based on the pods shared with other SPs, the SP owning the fat-tree allocates bandwidth resources across the links that provide access to such pods, and the servers within the pods.
4) SPs choose how many bandwidth is allocated in the links that provide access to their gateway nodes, that is the entry points to the fat-trees they own.

Right after the sharing of resources, the simulator creates a graph for every SP. This graph is a partial view of the infrastructure that each SP can access, and it contains the SP's fat tree, the links and gateway nodes that it can use to access other SPs' infrastructure, and the pods that other SPs have shared to grant access to their servers.

Fig. 1 depicts the graph that SP1 has. In this case only two other SPs share their fat-trees, and they give access to 1 and 2 pods, respectively.

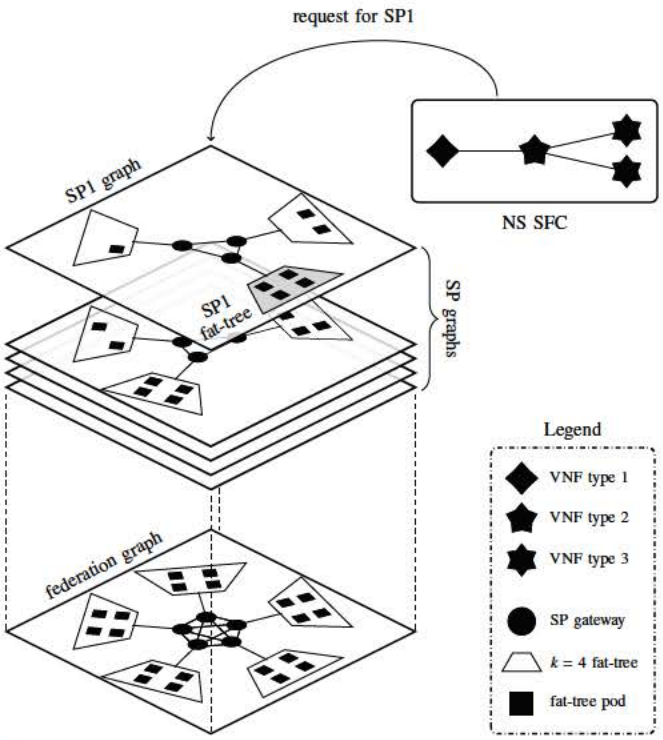



Fig. 1: Illustration of a federated multi-domain infrastructure of 5 SPs and their respective graphs. A NS mapping request arrives to SP1.

## IV. Mapping algorithms

This section presents the mapping algorithms used to solve the VNF mapping problem in a multi-domain environment.

All our algorithms are variations of a greedy approach that tries to minimize the overall delay of the mapped NS (1), and to do so it iterates through the SFC's VNFs trying to find the closest server that can host them attending computational constraints (4). The algorithms ensure that delay and bandwidth constraints are satisfied as well for the mapped NS — equations (2) and (3), respectively.

$$minimize \sum_{NS \in requests} delay_{map}(V_1, V_l) \tag{1}$$

$$s.t. \quad delay_{map}(V_i, V_k) \leq delay_{req}(V_i, V_k) \tag{2}$$

$$\sum_{NS} \sum_{(V_A,V_B) \in NS} u_{l,(V_A,V_B)} \cdot bw(V_A, V_B) \leq bw(l), \forall l \tag{3}$$

$$\sum_{NS} \sum_{V \in NS} u_{s,V} \cdot comput_V \leq comput_s, \quad \forall s \tag{4}$$

In the optimization problem solved with our algorithms, (1) to (4), $V_1$ and $V_l$ are the first and last VNFs in the NS. The variables $u_{l,(V_A,V_B)}$ and $u_{s,V}$ are booleans used to know if the link $l$ is used to connect VNFs $V_A$ and $V_B$, and if $V$ is mapped in server $s$. To describe the restrictions we use variables to express the bandwidth available in a link $bw(l)$ and the required to connect two VNFs $bw(V_A, V_B)$; on the other hand

$comput_V$ and $comput_s$ hold the computational requirements (CPU, memory and disk) required by a VNF $V$, and the ones present in a server $s$. And last but not less important, $delay_{map}$ and $delay_{req}$ are used for the delay between 2 mapped VNFs, and for the required delay between 2 VNFs in the NS mapping request.

In the following subsections we present all the algorithms.

### A. Greedy algorithm

The implemented greedy algorithm (Algorithm 1) iterates in order through every VNF in the SFC of the NS request. It starts looking for the servers that have enough CPU, memory and disk to host a VNF (line 1.4). Then it traverses the graph starting from the server where the previous VNF was mapped until it finds another server capable of hosting the next VNF (line 1.5). Once the VNF is mapped, the algorithm extracts the following VNFs directly connected to the one already mapped — this is what we call the neighbor VNFs — and repeats the process of finding the appropriate servers to host them, and traversing the graph to find a path between them and the server where the previous VNF was mapped.

To keep track of the consumed resources in the placement, a resource watchdog is responsible of allocating the bandwidth used along the paths and the server resources that every VNF requires (line 1.10). The resource watchdog is not asked to allocate resources in case there are no capable servers of hosting the VNF, or there is no path with available bandwidth to connect it with the previous one. In case the algorithm fails, the watchdog frees resources previously allocated, and it exits with an error.

During the mapping process a *NsMapping* object is created to keep track of the paths found between the servers selected to host the VNFs of the SFC.

**Algorithm 1** Greedy search

```
1:  function GreedyMapping(NsReq)
2:      for VNF in NSreq do
3:          for nextVNF in neighbors(VNF) do
4:              capServs = capableServers(opView, nextVNF)
5:              path = findPath(VNFserv, nextVNF, capServs)
6:              if no path then
7:                  watchDog.unwatch()
8:                  return ERROR
9:              else
10:                 watchDog.watch(VNFserv, nextVNF, path)
11:                 NsMapping.setPath(VNF, nextVNF, path)
12:                 NsMapping.setLnkDelay(VNF, nextVNF,
                                          path.delay)
13:             end if
14:         end for
15:     end for
16:     return NsMapping
17: end function
```

### B. The findPath method

In this subsection we present the different implementations of the *findPath* method. This method is used to search a server capable of hosting a VNF ensuring that delay and bandwidth requirements are satisfied. Implementations as the random walk, Dijkstra and BFS have already been studied; but not the DFS that we propose in this work.

*1) Random walk:* This algorithm traverses the SP available resources graph choosing randomly the links to visit (all the links have same possibilities of being visited). The implementation includes a hash table of already visited nodes to avoid choosing them twice in the random walk.

*2) Dijkstra:* We adapt Dijkstra's shortest path algorithm to our scenario. The modified version discards links and paths that don't satisfy the bandwidth and link restrictions imposed by the NS. It uses link delay as edge cost to reach the server nodes that can host the VNF to be mapped.

*3) BFS:* Another alternative is to search a server that can host a VNF using the BFS algorithm. This implementation creates a tree having as root the previous VNF and expands the tree in a BFS manner across the SP graph.

*4) DFS:* This implementation traverses the SP graph using a tree but following the DFS strategy. It has not been tried in the literature yet, but it reaches the servers of the SP graph faster than the previous implementations. This is because it goes directly to the leaf nodes of the generated tree.

### C. BFS and DFS run-time complexity

If all nodes and paths in the SP graph are visited to map a VNF, BFS and DFS deal with their worst case scenario:

$$O\left((k-1)^6 \cdot \left[\left(\frac{k}{2}\right)^2 - 1 + (p-1)\right]^2\right) \tag{5}$$

(5) run-time complexity refers to the paths that DFS and BFS do in the worst case. $k$ is the fat-tree degree, and $p$ is the number of SP gateways in the meshed scenario (in Fig. 1 we have $p = 5$ gateways in the federation graph). Every core, aggregate and edge switch of a fat tree has $k$ links, and we need to walk through three switches (one of each type) to reach the gateways that connects the initial SP to the others. Each gateway has connection to the $\left(\frac{k}{2}\right)^2$ core switches underneath, but it also has links with another $p-1$ SPs within the federated scenario of this work.

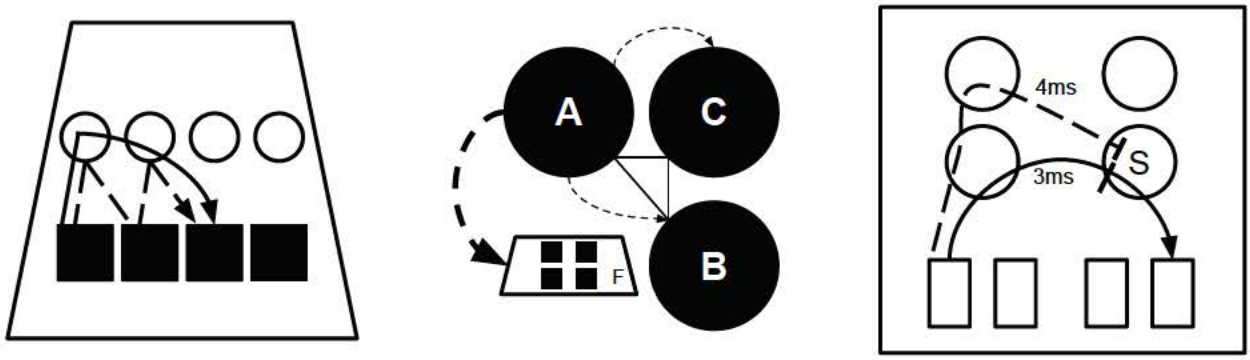
We can express (5) in terms of the number of nodes $N$ (switches, gateways and servers) of a fat-tree [13]:

$$O\left(N^{\frac{1}{3}}N^3\right. \tag{6}$$

### D. BFS and DFS cutoffs

In the following lines que describe the "cutoffs" introduced to improve the worst case run-time complexity (6):

Forbidden moves: the algorithm can not go from one server to another of the fat-tree if it does not use the shortest path.




(a) **Forbidden move** as a dashed line, shortest move across pods as continuous line. Circles are switches, boxes are pods.

(b) $P(A,F) > P(A,C) = P(A,B)$ thanks to **DFS priorities**. Circles are SP gateways and the trapezoid is a fat-tree.

(c) The dashed path reaches switch S with more delay than the continuous line **already visited** path.

Fig. 2: Cutoffs representation

DFS priorities: when DFS is visiting a gateway node, it must visit fat-tree underneath before other gateway nodes.

Already visited: if the algorithm is in a node that was previously visited through a path that reached it traversing links with lower delays, the node is not visited.

The first two cutoffs are based in infrastructure knowledge. This is novel in the VNF mapping algorithms context and it is one of our contributions to the state of the art.

### *E. BFS and DFS run-time complexity with cutoffs*

With the described cutoffs of Section IV-D the search space is reduced considerably. With this improvement, BFS is close to be a kind of Dijkstra search, because it checks if nodes have already been visited, and if the new path to reach already visited ones improves the cost of a previous path. But it has the advantage of the forbidden moves, and that makes the algorithm going forward to server nodes using the shortest paths that fat-trees were designed for [13]. This means that movements like going through an intermediate pod to reach a third one where the target server is located, are avoided. With DFS the speedup is even better, since in 9 comparisons it can reach a server from another SP. Then it goes straightforward to the servers, and in case it chooses properly the first server to visit, it goes faster than BFS with the implemented cutoffs. Having the cutoffs implies comparisons in every node the algorithm visits. In the switches this means having $k$ comparisons to each neighbor, and these comparisons are $O(1)$ since they rely on math operations or hash tables checks. Then, in the scenario where every already visited node has always been reached by a better path, BFS and DFS will visit each node of the fat tree only once and make $k$ comparisons in all of them. So the best scenario thanks to the cutoff improvements has a running time complexity of:

$$O(N) \tag{7}$$

Due to paper length limitations, we cannot prove that this best case scenario is very common in fat-trees that have links of same characteristics.

### *F. Tabu search algorithm*

It is well known that a greedy strategy leads to suboptimal solutions in the mapping of VNFs, hence we try to improve the solutions using a meta heuristic called tabu search. This algorithm is based on an initial solution, that in our case is provided by the greedy algorithm of section IV-A using all different implementations of the *findPath* method of section IV-B.

When the tabu search receives an initial solution it repeats the following operations in order:

1) Select the next VNF inside the NS.
2) Mark the server where it is mapped as tabu for $t$ turns.
3) Execute *findPath* to search a new server to host it that is not marked as tabu.
4) Repeat steps 1)-3) through all the VNFs inside the NS.
5) Check if the new mappings have decreased the NS end-to-end delay. If it is the case, store the solution.
6) Decrease counter $t$ for all tabu servers and back to 1).

These steps force to find new solutions hoping that the end-to-end delay of the mapped NS will be reduced. The idea is trying to go out of local minimums.

## V. Stress test

In this section we carry out an experiment in which we perform NS mapping requests in the federation graph as the resources are reduced. We try several algorithms to test their performance not only under circumstances when all resources are available, but when they deal with "stress", understanding it as lack of resources to do the mapping.

For the simulations explained in this section we have used a Dell PowerEdge C6220 with 2 Intel Xeon E5-2670 @ 2.60GHz processors and 96GB of memory. To schedule the

TABLE I: Tabu search best iterations and blockings parameters settings to reach highest acceptance ratio.

| algorithm | iterations | blockings | avg. time | acceptance (%) |
|---|---|---|---|---|
| DFS | 6 | 4 | 3.24 sec. | 61.5 % |
| BFS | 6 | 2 | 5.04 sec. | 63.5 % |
| Dijkstra | 5 | 3 | 4.37 sec. | 64% |

TABLE II: Acceptance ratios as resources are reduced

| | 0/10 | 2/10 | 4/10 | 6/10 | 8/10 | 10/10 |
|---|---|---|---|---|---|---|
| **DFS** | 100% | 90.75% | 61.25% | 27.25% | 3% | 0% |
| **BFS** | 100% | 89% | 59.75% | 27% | 3% | 0% |
| **Dijkstra** | 100% | 89.75% | 60.5% | 27.75% | 2.75% | 0% |
| **tabu** | 100% | 89.75% | 61.75% | 28.25% | 3.25% | 0% |

simulation jobs in parallel we have made use of GNU parallel [14].

We use python to implement the algorithms of Section IV, and we rely on the NetworkX software package [15] to manipulate the graphs. All the code used to obtain the results presented in this paper is published as a public repository[1].

### A. Experiment setup

The graphs generated for this experiment are made up of 20 SPs, each one has a $k = 4$ fat-tree data center connected to the federation. The gateway nodes are connected as a full mesh. In terms of resources sharing, every SP has access to the computational resources of other 9 SPs, and the foreign SP can share up to 4 pods with it. Every server equally shares its resources with the SPs that can access itself.

In the experiment we launch 400 NS requests (each of them made up of 6 VNFs), having every VNF same computational resources requirements, an end-to-end SFC delay of 15 time units, and links requesting 1 bandwidth unit. Each request is performed by one of the 20 SPs, to decide which one requests the NS we use a random variable that follows a uniform distribution $SP \sim \mathcal{U}\{1, 20\}$.

For the initial step of the experiment all of the 400 NS requests must have enough resources to be allocated, that is a 100% acceptance ratio. To achieve it we start with the following conditions: 1 time unit of delay and 2400 bandwidth units in the links used to connect switches, gateways and servers in the generated infrastructure; and computational resources in each server (320 present in the multi-domain graph of this experiment) to host up to 1 VNF for every SP that can access it (remember every requested VNF has same computational requirements in the stress test).

### B. Tabu search parameters

Before performing the stress test we checked which implementation of the *findPath* must be used in the initial greedy algorithm that tabu search uses as initial solution to perform modifications. We also tuned the tabu search parameters to get the best acceptance ratios. Unlike Section V-A, we modified the incoming NS request requirements so the link's delay and bandwidth, and VNF computational resources are not always the same. If the VNFs to be mapped require between 1/200 and 1/20 of a single server disk resources, between 1/200 and 1/50 of the CPU resources, and between 1/200 and 1/12 of the server memory resources; then the generated multi-domain is not able of hosting all the incoming NS requests.

[1]https://github.com/MartinPJorge/vnfs-mapping/commit/c8172327860443ac8abcc9f4a51d66abf5c26e19

With this in mind we performed 400 NS requests across the 20 SPs using Dijkstra, DFS, and BFS (the last two with the cutoffs) as the *findPath* algorithms to be used in the initial solution provided to the tabu algorithm. Then we seek how many iterations the tabu meta heuristic must perform over the SFC trying to remap every VNF, and for how many iterations the performed mappings must be blocked (marked as tabu). Table I shows the best configurations and average mapping time per NS request among the 400 ones, and the acceptance ratio. According to the table, Dijkstra achieves the best acceptance ratio, but the DFS gets only a 2.5% lower acceptance ratio while obtaining the quickest average mapping times.

### C. Resource reduction stress test

For this section's experiment the first step is to have a 100% acceptance ratio scenario as the described in Section V-A, then computational resources are reduced in steps of tenths until every server in the infrastructure has no more CPU, memory and disk available.

After decreasing the computational resources of all the servers to a tenth, our simulator performs the 400 NS requests with the requirements of Section V-A. It tries with 4 different algorithmic approaches. Three of them are just greedy search using Dijkstra, BFS and DFS with cutoffs for the *findPath* method. The tabu algorithm tested is the one with the parameters that retrieved the best acceptance ratio among the ones tried in Section V-B (the one based in Dijkstra). The random walk implementation of the *findPath* method is not included in the experiment because it does not have 100% acceptance ratio even when there are enough computational and bandwidth resources for the incoming 400 NS requests of the experiment setup.

All the tried algorithms yield acceptance ratios that differ from each others in $\leq$ 2.24% in every step reducing the resources (see Table II). That is, the experiment reduces a 10% the computational resources, and in each step all the four algorithms mentioned in the previous paragraph obtain very similar acceptance ratios in the 400 NS requests they are asked to map.

But although acceptance ratios are almost the same, the running times differ within the four used algorithms (see Fig. 3). The highest execution time has been reached in the tabu search, while the greedy search using DFS to find server nodes is the one that has taken less time to perform the 400 NS request mappings. The reason why tabu decreases its execution times as resources are reduced, is because acceptance ratio

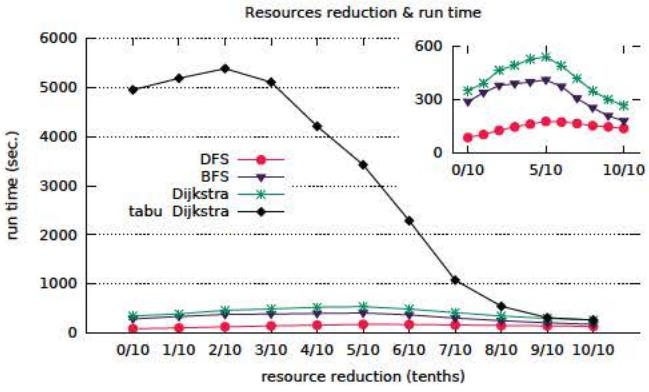



Fig. 3: Running time of 400 NS mapping requests using different algorithms as server resources are reduced.

is diminished, and if the initial greedy search fails, the tabu algorithm exits.

In the top-right corner of Fig. 3, a zoom is applied to the graph to display the differences between the greedy algorithms using DFS, BFS and Dijkstra to reach shortest paths. The greedy DFS is the quickest performing the mapping of the 400 NS requests in every resource reduction scenario.

## VI. Conclusions

This paper presents our work on VNF mapping algorithms in multi-domain environments where several SPs share resources in a federation. We explain the graphs that our simulator generates to model such federated scenario, and how the SPs share the bandwidth and computational resources in the fat-trees that we use to model domains' data centers.

On top of the generated multi-domain graphs we have tested several algorithms based on a greedy search, and we have tried to improve the solutions obtained with that approach by using a meta heuristic tabu search.

The contribution of our work is the usage of DFS to traverse the multi-domain graphs, the modification of already existing tabu search presented in [9] so the SFC's delay is taken into account, and the cutoffs applied to DFS and BFS to boost up the algorithms performance relying on the fat-tree properties. Such cutoffs make possible reaching $O(N)$ running times in very likely scenarios in the fat-tree topologies.

Our work shows that among the implemented algorithms, most of them obtain very similar acceptance ratios. Indeed, although we have implemented the tabu search trying to increase the acceptance ratio, results have shown that these little improvements lead to a high overhead in the running times, and the best of the implemented solutions (trading off execution time and acceptance ratio) is the DFS we propose with the incorporation of the cutoffs.

As future work we want to try the cutoff considerations in the Dijkstra algorithm, and use Fibonacci heaps to store the unvisited nodes. We also plan to compare our solutions with the optimal ones that ILP solutions can retrieve. Finally we would like to model with probability how likely it is to achieve our $O(N)$ best scenario, implement another well known meta heuristic algorithms like the simulated annealing, and try nature-based collaborative algorithms.

## Acknowledgment

This work has been partially supported by the EU H2020 5G Exchange (5GEx) innovation project (grant no. 671636) and by EU H2020 5G-Transformer Project (grant no. 761536).

## References

[1] X. Li and C. Qian, "The virtual network function placement problem," in *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, April 2015, pp. 69–70.

[2] Z. Allybokus, N. Perrot, J. Leguay, L. Maggi, and E. Gourdin, "Virtual Function Placement for Service Chaining with Partial Orders and Anti-Affinity Rules," *arXiv preprint arXiv:1705.10554*, 2017.

[3] V. S. Reddy, A. Baumgartner, and T. Bauschert, "Robust embedding of VNF/service chains with delay bounds," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Nov 2016, pp. 93–99.

[4] C. Ghribi, M. Mechtri, and D. Zeghlache, "A Dynamic Programming Algorithm for Joint VNF Placement and Chaining," in *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking*, ser. CAN '16. New York, NY, USA: ACM, 2016, pp. 19–24. [Online]. Available: http://doi.acm.org/10.1145/3010079.3010083

[5] A. Hirwe and K. Kataoka, "LightChain: A lightweight optimisation of VNF placement for service chaining in NFV," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, June 2016, pp. 33–37.

[6] M. Mechtri, C. Ghribi, and D. Zeghlache, "A Scalable Algorithm for the Placement of Service Function Chains," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 533–546, Sept 2016.

[7] F. Carpio, S. Dhahri, and A. Jukan, "VNF Placement with Replication for Load Balancing in NFV Networks," *arXiv preprint arXiv:1610.08266*, 2016.

[8] M. A. Lopez, D. M. F. Mattos, and O. C. M. B. Duarte, "Evaluating allocation heuristics for an efficient virtual Network Function chaining," in *2016 7th International Conference on the Network of the Future (NOF)*, Nov 2016, pp. 1–5.

[9] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and S. Davy, "Design and evaluation of algorithms for mapping and scheduling of virtual network functions," in *Network Softwarization (NetSoft), 2015 1st IEEE Conference on.* IEEE, 2015, pp. 1–9.

[10] C. J. Bernardos, B. P. Gerö, M. Di Girolamo, A. Kern, B. Martini, and I. Vaishnavi, "5GEx: realising a Europe-wide multi-domain framework for software-defined infrastructures," *Transactions on Emerging Telecommunications Technologies*, vol. 27, no. 9, pp. 1271–1280, 2016.

[11] B. Németh, J. Czentye, G. Vaszkun, L. Csikor, and B. Sonkoly, "Customizable real-time service graph mapping algorithm in carrier grade networks," in *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on.* IEEE, 2015, pp. 28–30.

[12] B. Németh, B. Sonkoly, M. Rost, and S. Schmid, "Efficient service graph embedding: A practical approach," in *Network Function Virtualization and Software Defined Networks (NFV-SDN), IEEE Conference on.* IEEE, 2016, pp. 19–25.

[13] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 63–74, Aug. 2008. [Online]. Available: http://doi.acm.org/10.1145/1402946.1402967

[14] O. Tange, "GNU Parallel - The Command-Line Power Tool," *;login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, Feb 2011. [Online]. Available: http://www.gnu.org/s/parallel

[15] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring Network Structure, Dynamics, and Function using NetworkX," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.